

# A Wiki for Mizar: Motivation, Considerations, and Initial Prototype<sup>\*</sup>

Josef Urban<sup>1\*\*</sup>, Jesse Alama<sup>2\*\*\*</sup>, Piotr Rudnicki<sup>3†</sup>, and Herman Geuvers<sup>1</sup>

<sup>1</sup> Radboud University, Nijmegen, the Netherlands

<sup>2</sup> New University of Lisbon, Portugal, Portugal

<sup>3</sup> University of Alberta, Edmonton, Canada

**Abstract.** Formal mathematics has so far not taken full advantage of ideas from collaborative tools such as wikis and distributed version control systems (DVCS). We argue that the field could profit from such tools, serving both newcomers and experts alike. We describe a preliminary system for such collaborative development based on the Git DVCS. We focus, initially, on the Mizar system and its library of formalized mathematics.

## 1 Introduction and Motivation

Formal mathematics is becoming increasingly well-known, used, and experimented with [9, 10]. Verification of major mathematical theorems such as the Kepler Conjecture [8], the Four Color Theorem [7], and the increasing use of verification for critical software and hardware [11, 5] are pushing the development of interactive verification tools. Indeed, there are already various online repositories of formal proofs [1, 15, 4] and catalogs of deductive tools [21].

The goal of the work presented here is to make formal mathematics *accessible online* to interested parties by making the subject widely available through online tools and interfaces. We are particularly interested in providing fast *server-based* tools for verification, web presentation, and collaborative development of formal mathematics.

We believe that such tools are important for our field to attract newcomers: they provide an attractive environment for exploring the world of formal reasoning and the tools of the trade. The technology we have in mind is vital also for

---

<sup>\*</sup> The final publication of this paper is available at [www.springerlink.com](http://www.springerlink.com)

<sup>\*\*</sup> Supported by the NWO project *MathWiki: A Web-based Collaborative Authoring Environment for Formal Proofs*.

<sup>\*\*\*</sup> Partially supported by the European Science Foundation research project *Dialogical Foundations of Semantics* within the ESF EUROCORES program *LogICCC: Logic in the Humanities, Social and Computational Sciences* (funded by the Portuguese Science Foundation, FCT LogICCC/0001/2007).

<sup>†</sup> Partially supported by NSERC.

existing members of the formal mathematics community simply by making systems for formal mathematics easier to access and use. Ideally, the system should be so straightforward that, to satisfy some momentary curiosity about a formalism or a proof or a theorem, one could just visit a web page instead of suffering through the installation of a new system. In the long run, we foresee a web-based repository of mathematics realizing the vision of the QED Manifesto [20].

Our effort has three principal features: it

- is based on the notion of a wiki (understood here as a support for distributed, web-based collaborative project)
- uses distributed version control system(s), and
- uses server-based software to do the “heavy lifting” of verification and proof checking.

Let us briefly characterize these features of our approach.

The first main feature of our approach is the use of wikis. Wikis are well known; they offer collaborative web-based information resources. The original wiki [2] is nearly 15 years old. This new genre of web site has grown enormously. Wikipedia is, evidently, the most prominent example of a wiki: even casual Internet users are not only aware of Wikipedia but use it often: links to Wikipedia pages are often among the top results of web searches.

Mathematics is, thankfully, no exception to this trend. Wikis and other online repositories and communities for mathematics abound: arXiv, MathOverflow [12], T. Gowers’ PolyMath [18], Wolfram MathWorld [13], PlanetMath [17], ProofWiki [19], etc.<sup>4</sup> These constitute just a sample of mathematics’ web presence; it does not seem feasible to give a complete survey of the subject. Yet, at present there are no mathematical web sites for *formal* mathematics that permit collaborative editing and other online tools (though we know of one prototypical effort [3]). We aim to fill this gap.

The second main feature of our approach is the use of distributed version control systems (DVCS). Such systems are eminently suitable for large collaborative efforts: they allow one to maintain repositories of texts or code in the presence of multiple people working independently or cooperatively. DVCSs are becoming more widely used, both in the commercial sector, in academia, and in the free software community. Our approach is novel because the application of DVCSs to maintaining large bodies of formal mathematical proofs—objects that are both computer code and human-readable text—is largely underdeveloped.

The third and final main feature of our approach is the use of server-based tools. Like DVCSs and wikis, such systems are becoming widely used. The general reason for this is that the Internet is becoming ubiquitous, faster, and more reliable. Server-based approaches can arguably help to attract newcomers to the subject because it spares them the burden of locally installing formal mathematics software and its associated libraries. Moreover, computations that one might want to carry out are sometimes large and should (or must) be carried out on servers rather than less powerful client hardware. In our case, proof checking and

---

<sup>4</sup> Note that majority of these on-line services are non-profit efforts.

the generation of rich semantic HTML presentations of formal proofs is, often, quite expensive.

The rest of the paper is organized as follows:

- Section 2 briefly describes the primary applications of a wiki for formal mathematics.
- Section 3 lists the essential features of DVCSs, and how we use them to provide robust and flexible back-end for our formal mathematical wiki.
- Section 4 briefly discusses the current methods for developing the *Mizar* Mathematical Library (MML) and identifies the main bottlenecks of the current model for massive distributed formalization efforts.
- In Section 5 we discuss the features that a formal mathematical wiki should have and the requirements it should satisfy; we focus on the problem of maintaining the correctness of a formal library.
- The initial implementation of the wiki for *Mizar*, based on the *Git* DVCS, is given in Section 6. We explain the *Git* repository structure, the communication and division of work between the different repositories and branches, how we use *Git* hooks for guarding and updating the repositories, how we extract and recompute dependencies from the changed library and its re-verification and regeneration of its HTML presentation.
- Possible future extensions are discussed in Section 7.

## 2 Use Cases

We intend to provide tools for various collaborative tasks in the formal mathematics communities. Our efforts should also aim to include the whole mathematical community and be beneficial to it: formal mathematics is a natural extension of traditional mathematics in which the gaps, metaphors, and inaccuracies are made “computer-understandable” and thus, in principle, mechanically verifiable. The major use cases that we have in mind are

- public browsing of a human-readable web presentation of computer-verified mathematics;
- facilitating the entrance to the formal mathematics community;
- library refactoring (from small rearrangements of individual items and minor changes, to major overhauls);
- authoring contributions, both large and small;
- supporting the design of a formalization structure (concept formation, fleshing out definitions, bookkeeping postulates and conjectures);
- offering tools to get help with simple proof steps;
- gradually formalizing informal texts;
- translating contributions between/among proof assistants, archives, and natural languages;
- merging independently developed libraries.

In this paper, we narrow our focus to examine how some of the above tasks apply to the *Mizar* system. Thus, we are interested in the aspects of the above problem that arise when working within a single, fixed formal framework. However, we keep in mind the rather grand scale of the issues at hand.

### 3 Towards distributed collaboration

One of the most exciting but apparently unexplored topics in formal mathematics is the use of distributed version control systems (DVCSs). Such systems support tracking the development of work among authors whose paths can be non-linear, proceeding down a variety of routes/ideas that may or may not converge upon a final, conventionally agreed-upon goal.

When thinking about DVCSs, a potential misunderstanding can arise that we should correct. One might get the impression that DVCSs simply encourage chaos: without a central repository, who is to say what is correct and what is not? And would not DVCSs lead to a fragmented community and wasted labor?

Although we can conceive such dystopian possibilities, we prefer another point of view. We want to emphasize the *organized* in *organized chaos*, rather than the troubling noun. It is helpful to think of DVCSs not as a source of chaos, but rather as a tool for putting some structure on the distributed efforts of a group of people sharing a common interest. In practice, DVCSs are used to organize the efforts of many people working on crucial projects, such as the Linux kernel<sup>5</sup>. Although the DVCS model does not require a central, standard repository to which everyone refers, there often are strong conventions that prevent the disarray and confusion that can come to mind when one first hears about DVCSs. In the case of the Linux kernel, for example, the entire community practically revolves around the efforts of a single person—Linus Torvalds—and a rather small group of highly trusted programmers. A fairly wide number of people still contribute to the Linux kernel, but in the end, essentially all proposed contributions, if they are accepted at all, pass through the core developers. At least for the foreseeable future we propose to follow the same approach: a handful of experienced Mizarians<sup>6</sup> will decide what constitutes the current official release of the distributively developed system.

Foremost, we need to ensure that the current practices of developing the Mizar Mathematical Library (MML), which evolved over a number of years, can still be supported. Indeed, the core Mizar developers can easily continue their current practices using DVCSs. We would like to switch to distributed development in an incremental fashion and we foresee deep involvement of the current Mizar developers while switching to the new technology.

For our first pass at a formal mathematics collaborative tool, we have chosen the Git system [6]. Git is originally developed by Linus Torvalds, the original author and primary maintainer of the Linux kernel, for working with the large number of contributions made to the Linux kernel. Git enjoys widespread use in various software communities (commercial, academic, and open-source).

Our choice of Git is, to some extent, arbitrary. A number of widely used DVCSs are available (*monotone*, *bzr*, *arch*, *mercurial*) that could support the collaborative tasks we envision for a formal mathematics wiki. (Indeed, one project

---

<sup>5</sup> Just to give some feeling for the size, the compressed sources of the Linux kernel are roughly 53 MB; the compressed sources of the proofs in the Mizar Mathematical Library are 14 MB.

<sup>6</sup> We thank Yuji Sakai for stressing the charm of this name.

with aims similar to ours—*vdash* [25]—has proposed to use *monotone*.) We cannot, therefore, robustly defend our choice of *Git* except to say that *some* choice of DVCS must be made, and any other choice of DVCS would likely be just as arbitrary. We choose *Git* because of our familiarity with it, and its wide usage in many projects that we know of.

A full description of *Git* can be found on its homepage [6] and the system can be learned through a number of tutorials prepared by the *Git* community. We believe that *Git*’s concepts and operations are a good match for the kinds of collaborative tasks that a formal mathematics wiki should support. Here we limit ourselves to a skeletal presentation of *Git*’s main features.

Like other version control systems, the *Git* system is based on the notion of a *repository*, a structured collection of objects (texts, computer code, documentation, etc.) being worked on by members of a team. As a developer makes changes to the repository, a sequence of *commits* are made. The sequence can split at various stages, as when the developer or the community wish to take on a subproject or otherwise pursue some line of investigation. The full history of a repository can thus be more aptly understood as a tree rather than a simple sequence of changes.

A single developer can either start a new project afresh, or can *clone* the repository of some other developer working on the project of interest. Unlike traditional non-distributed version control systems such as *rcs*, *cvs* and *subversion*, *Git* repositories (and those of some other DVCSs, too) are “complete” in the sense that the developer has essentially unlimited access, even when offline, to the entire history of a repository. When online, the developer can share his changes with others by *pushing* up his changes to another repository (or repositories), and he can stay connected to his colleagues by *pulling* down their changes.

The *Git* system also provides *hooks* that can be used to implement various guards and notifications during the various repository actions (*commit*, *push*, etc.). This can be used to allow only certain kind of changes in the repository. For example, our initial wiki prototype for Mizar (See Section 6) makes use of suitable hooks to ensure that the updated articles and the whole updated library repository are always in a correct state.

In our initial prototype, we introduce a publicly accessible, central repository—in the spirit of Wikipedia or the main Linux kernel branch—that serves as a correct and verified snapshot of the current MML. Our tools are targeted at public editing of the repository, while ensuring the library’s coherence<sup>7</sup> and correctness as the changes are made.

## 4 A special case: the Mizar developers

Among the first targets for our implementation are the core Mizar developers. It will be worthwhile, then, to explain how Mizar is currently developed.

---

<sup>7</sup> Coherence of MML is closely related to formal correctness and is a narrower notion than integrity of MML, cf. [22].

The history of the **Mizar** project is briefly presented in [14] and a brief overview of the current state of the project can be found in [16].

The development of the **Mizar** Mathematical Library (MML), the principal, authoritative collection of **Mizar** contributions (called “articles”), has been the main activity of the **Mizar** project<sup>8</sup> since the late 1980s, as it has been believed within the project team that only substantial experience may help in improving (actually building) the system. An article is presented as a text-file and contains theorems and definitions. (At the time of writing, there are 1073 articles in MML, containing 49548 facts/theorems and 9487 definitions.) The articles of MML—in source form—are approximately 81MB of data.

The current development of MML is steered by the Association of **Mizar** Users (in Polish: Stowarzyszenie Użytkowników Mizara, abbreviated SUM), which owns the copyright to the library. SUM appoints the MML Library Committee—the body responsible for accepting articles, arranging reviews, and maintaining official releases of the library. MML is organized in an old-fashioned way, resembling printed repositories of mathematical papers. Since MML articles exist electronically, it is relatively easy to re-factor the MML contents by, say, changing items such as definitions and theorems, or by deleting, or by moving them. After such modifications, the **Mizar** processor is run on the entire MML to verify its coherence. Currently, the Library Committee does the greatest amount of refactoring of MML, which includes the following activities:

- updating existing articles to take advantage of changes in the **Mizar** language, the **Mizar** processor, and changes in MML;
- moving library items where they more appropriate locations;
- building an encyclopedia of mathematics in **Mizar** which consists of creating new articles by moving topically related items from all over the MML into one place;
- generalizing definitions and theorems;
- removing redundant items; and
- merging different formalizations of the same material.

This process is under control of the Head of the Library Committee; about three people do most of the revisions. It can also happen that the original **Mizar** authors re-factor their own past contributions after they notice a better way to formalize the material. The remaining MML revisions are usually suggested by posting to the **Mizar** mailing lists, or mailing directly to someone on the Library Committee.

Information about what each of the MML refactorers is doing is not broadcast in any widely accessible way. The typical method is through an email announcement: “I am now formalizing XYZ, so please wait for my changes”. Such announcements are frequently accompanied by a solicitation for remarks and discussion.

Such a process for collaboratively editing MML can be problematic and is far from ideal. The main problem with the present method is that it does not scale

---

<sup>8</sup> <http://mizar.org>

up: although it can be used with some success among few people, we imagine a much larger community where the collaboration problem surely cannot be effectively solved by the present approach. Moreover, new users face an unpleasant experience of developing a new article (or a group of interrelated articles) and, before finishing their work, they learn of a newer, substantially different official MML version that is (partly) incompatible with their working articles. Updating such an unfinished development can be quite time-consuming (but the Library Committee offers help in such cases).

## 5 Formal wiki features and issues

The experience of Mizar authors is that while developing some new formalizations they notice that many MML items can be improved by, say, weakening the premise of an existing theorem or widening the type of a parameter in a definition. Ideally, an author should be able to introduce such small changes to the repository instead of asking the Library Committee and waiting for a new version of MML to be released. With a DVCS such small changes can be incorporated into a daily build as there are mechanical means for verifying the correctness of such edits.

For the benefit of users, there must be something like an official MML version. We believe that with the DVCS development model, such official versions can be produced less frequently, subsequent official versions could differ more substantially, and all differences could be documented.

We foresee the following goals when treating MML as a wiki:

- content *correctness*;
- *incremental* editing; and
- *unified presentation* of the library for browsing.

Finding the right trade-off among these goals is not trivial. Allowing incremental editing of a formal text can lead to its incorrectness. On the other hand incrementality is an important feature of wikis. Thus the formal wiki should be able to save one's work even if it is a major rewrite of the existing code, and not completely correct. But the library is also a source of information for other users/readers and its current stable/correct version should be well presented. This means, however, that all kinds of unfinished work (active works-in-progress, abandoned attempts) should be reasonably hidden from the readers who just came looking for information.

Thus it seems that we have to combine the methods used for distributed software development—allowing many people to work on different parts, possibly breaking things, and easily accessing and merging each other's work—with the methods and mechanisms used for development of informal wikis like Wikipedia, where the content is primarily intended for human reading, and things cannot be “formally broken” too much (obviously, large projects like Wikipedia do have some automated structural checks, but evidently most of the review is done by humans). As of now we are only collecting experience on how to build and maintain a wiki of contents with formally defined semantics and correctness. It

is not clear to what degree issues raised in designs of less formal repositories are relevant to our task.

### 5.1 Degrees of Formal Correctness and Coherence

The important difference between text-based wikis like Wikipedia and formal libraries like MML is the strong, well-defined notion of *formal correctness*. As regards MML, one can consider several aspects/scopes of formal correctness that are not available (not even in principle) for traditional wikis such as Wikipedia.

To begin, consider the notion of *link coherence* on, say, Wikipedia. Suppose someone edits only the body text of the Wikipedia entry for the Eiffel Tower. The article’s internal coherence is unchanged, and the link coherence of Wikipedia as a whole is unaffected by the edit. However, other kinds of edits are permitted:

- if one deletes the Eiffel Tower article or renames it, then (part of) Wikipedia becomes link incoherent because some internal links become broken.
- if one deletes or renames a section of the Eiffel Tower article, then, as in the first case, it is quite possible that other Wikipedia articles become “tainted” by this edit by containing links to a subsection that no longer exists.

Internal link coherence can be enforced in various ways. Some methods are entirely computational (that is, can be carried out autonomously by a machine and require no human intervention); others depend in part on the (human) Wikipedia community. Further, this notion of coherence can be enforced by responding to potentially problematic edits, or by simply not giving users the means to make them. For example, the problem of deleting pages can be addressed by simply disallowing deletions. If such edits are allowed (perhaps only by certain users), an autonomous wikibot, constantly patrolling Wikipedia, can enforce internal link consistency by updating every Wikipedia article that refers to the Eiffel Tower article. This kind of bot-based approach could also respond to internal section renaming or deletion. Finally, internal link coherence can be enforced by Wikipedia’s editor’s tools, which give humans an environment for editing that does not straightforwardly permit such section renaming in the first place.

A wiki based on formal texts like Mizar articles permits one to define the notion of *change propagation*. When a user changes a piece of the Mizar library, we must take into account how this change affects other parts of MML. For example, if out of curiosity one changes the definition of the constant  $\tau = (1 + \sqrt{5})/2$  to, say,  $1/2$ , then, obviously, this edit will have some effect on the library (some theorems about Fibonacci numbers that involve this classical ratio will be invalidated). Likewise, if one rearranges the theorems or definitions in an article, this too can affect parts of the library. On the other hand, some edits are safe: one can append new content to the end of existing articles without affecting the library’s coherence.

The problem is that some modifications preserve the coherence of the wiki, but some do not. The typical task we face in maintaining the coherence of MML, considered as a wiki, is that a user has changed or added some articles to the



library, and we want to verify that these changes are admissible, that is, that they maintain the coherence of MML. In extreme cases, checking the admissibility of a user's edit can, conceivably, require re-verifying of the entire library (for example, imagine changing the the fundamental set-theoretical axioms on which MML rests). The cost of such admissibility checks is correspondingly large. However, there are a number of more or less typical cases that can be dealt with less expensively. In the next section we explain how we have implemented these checks.

## 6 Prototype

### Suitability of DVCSs and wikis

A natural object of interest when thinking about wikis and DVCSs together are wikis based on DVCS. According to a recent listing,<sup>9</sup> there are today more than a dozen wikis based on the Git DVCS or using Git as a back-end. Our design decision about wiki for Mizar is: the data (articles) in the Git repository should always be self-sufficient (independent of the wiki functionalities), and easily exchangeable between various installations (that can be used just for local development, as a back-end for another wiki-like system, etc.). Wikis form a dynamic field of interest, so tight coupling the Mizar data to a specialized and possibly not widely adopted wiki storage implementation could cause difficulties for developing alternatives and for integrating formal mathematics into other, larger frameworks. All such alternatives should provide convenient communication at the DVCS (data) level with others. This seems to be a reasonable invariant for building all kinds of web sites and tools for creation of formal mathematics (and probably for collaborative creation of code in general).

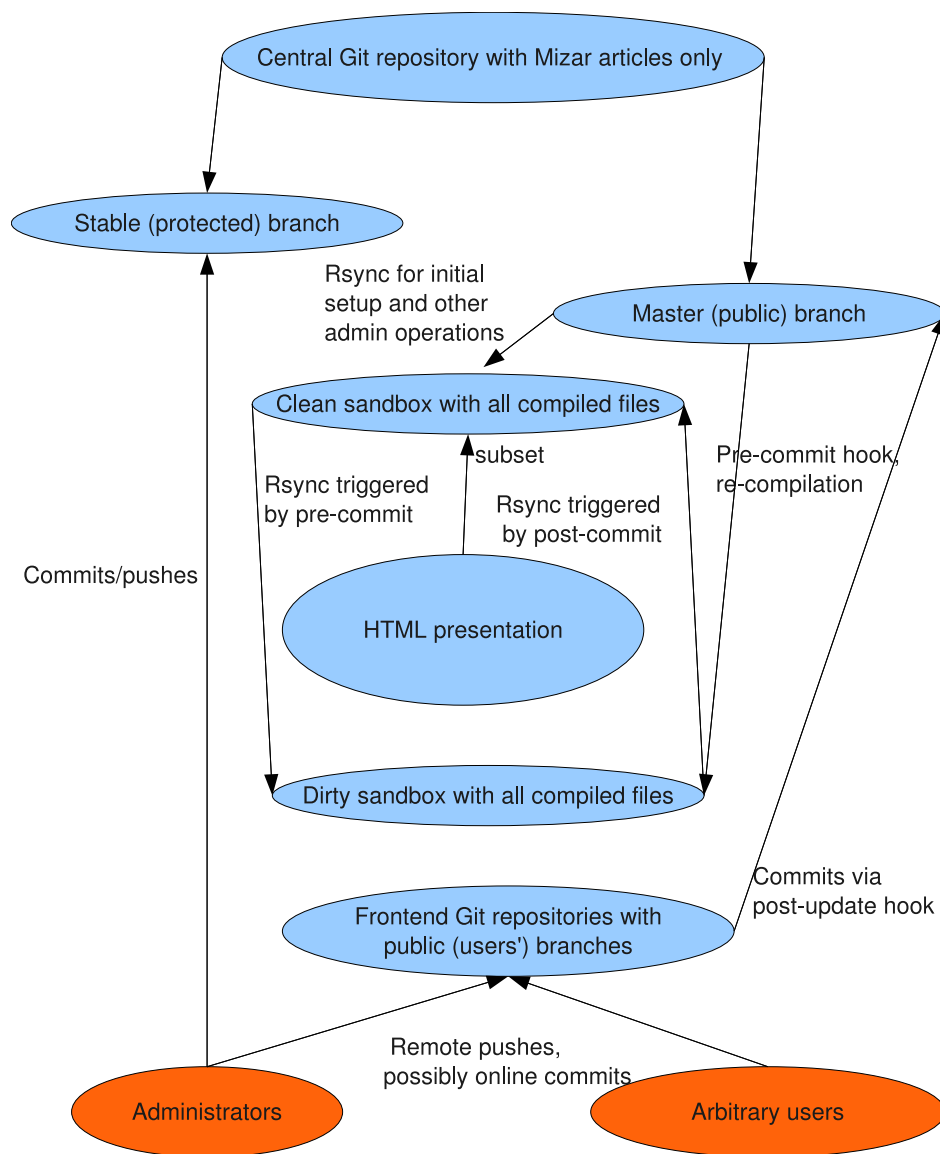
ikiwiki is one of the most developed and probably best-known *wiki compilers*. Wiki compilers process a file or set of files written in a special (usually simplified) syntax and turn them into static HTML pages. It is possible to build an initial prototype of a wiki for Mizar by customizing ikiwiki. We have explored this path and are aware of many functionalities of ikiwiki useful for our task. We have also considered the peculiarities that make our domain different from the typical (one-page-at-a-time) paradigm of such wiki compilers, and decided to gradually re-use and plug-in the interesting ikiwiki functionalities, rather than trying to directly customize the large ikiwiki codebase to our nonstandard task.

**Prototype Overview:** As we discussed earlier, our prototype (see Figure 1) is initially focused on the Mizar developers who edit their work mainly offline and submit their changes to be viewed online<sup>10</sup> by other developers. The repository structure (and suggested future features) of our prototype is as follows.<sup>11</sup>

<sup>9</sup> <http://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

<sup>10</sup> <http://mws.cs.ru.nl/mwiki>

<sup>11</sup> The code is available from <http://github.com/JUrban/mwiki>



**Mizar wiki structure**

**Fig. 1.** Mizar wiki structure

- There is a central repository on our server with the *stable* branch, and the *master* branch. This repository can be (using post-commit hook) kept in sync with a version published at, say, GitHub (<http://github.com>, a high-profile web site for collaborative Git-based development) or other easily accessible public Git repository web sites. Anybody can clone the published copy from such public locations, saving us some of the responsibility for maintaining access and sufficient bandwidth to our repositories.
- The repository contains basically only the source `.miz` files (Mizar formalizations), just as the Linux repositories only contain the source `.c/.h` files. All files that depend on the sources, and possibly intermediate meta-data constructible from them, are not part of the repository, and are excluded from the repository. We achieve this using Git's `.gitignore` feature.
- The *stable* branch of the repository is only available to administrators. New official versions of MML are pushed there.
- The *master* branch is the default one and it can be updated by any formally correct change (guarded by a suitable Git pre-commit hook). Essentially anybody can make changes (this is the wiki feature) using the mechanisms and checks described later. The *stable* branch and the Git history should serve as tools for easily reverting bad changes.
- The central repository is not directly updatable by users. Instead, it is cloned to another (frontend) repository on the server to which Git remote pushes can be made without any correctness checking. Limited control should be exercised to prevent malicious use<sup>12</sup>. Upon a successful remote push to the frontend, the Git post-receive hook is triggered. This hook attempts a commit to the master branch of the central repository, triggering in turn its pre-commit hook, and the formal verification of the updated library.
- Upon successful commit into the central repository, a post-commit hook is triggered. This hook generates HTML for the updated library, publishes it on the web, and does possible further related updates (updates of related clones on GitHub, notifications, etc.)
- The `gitweb` graphical web interface can be used for browsing the repository (comparing different changes and versions, and watching its history, etc.). Alternatives to `gitweb` abound: one is to use all the Git-related tools maintained and developed for GitHub.
- Editing is, initially, done locally, using a local customized editor like Emacs (used by most current Mizar users today); later, the articles are submitted using Git mechanisms. Similar “offline editing” is not uncommon in the wiki world: in the Emacs environment, for example, there are tools for similar offline interaction with Wikipedia. There are a number of options for providing additional direct in-browser editing (useful for smaller edits), ranging from the basic text-box and submit button to more advanced general web-based editors like in Wikipedia, to specialized online mathematical editors like ProofWeb.

<sup>12</sup> It is possible to have a number of such frontends, and with sufficient infrastructure in place to actually move the main non-verifying frontends again to public hubs like GitHub.

## 6.1 Prototype Implementation

The implementation of the pre-commit checks, the post-commit cleanup, and related infrastructure is based on the following. We make essential use of makefiles to specify which files need to be rebuilt, how to build them, and how they depend on other files. In the central repository we have one central makefile  $M$  and, for each article  $a$ , a makefile  $M_a$  that specifies on which other articles  $a$  depends (e.g., what notations, definitions, theorems  $a$  uses). The master makefile  $M$  has targets that ensure that the whole of the (about-to-be-submitted) MML is coherent and without errors. Naturally, when one submits a small change, it is generally not necessary to re-verify the entire MML, and the makefiles are designed to re-verify the minimal necessary set of dependencies. The verification is carried out in a “fully compiled” MML, so all the auxiliary results of previous verifications (analogous to `.o` files for GCC) are available, from which the make program can infer whether they need to be re-computed.

As the library changes, the dependencies among the articles can change, and re-writing the makefiles by hand each time a dependency changes would be tedious. Following tools like `makedepend` for C and other languages (and probably some similar tools for other proof assistants), we have created the `envget` tool based on the `Mizar` codebase for quickly gathering the dependencies that are explicitly declared by an article<sup>13</sup>. Thus the makefiles  $M_a$  for each article  $a$  are themselves generated automatically using makefile targets defined in the master makefile  $M$ . The XML output of `envget` is combined with a suitable XSL stylesheet, producing  $M_a$  for an article  $a$ , containing only one target, and specifying the articles on which  $a$  depends. These dependency makefiles are refreshed only when the corresponding article is changed. This leads to a reasonably efficient Makefile-based setup: only the dependencies of changed files get possibly changed, and if that happens, only the (dynamically changed) dependency sub-graph of MML influenced by the committed changes gets re-verified.

Where is the make program (governing verification of changes) invoked? How do we ensure that the central repository, assumed to be a coherent copy of the `Mizar` distribution, does not get “tainted” by incoherent user updates? In addition to a fully-compiled, coherent clean `Mizar` “sandbox” directory  $S_c$ , we maintain a (possibly) dirty sandbox directory  $S_d$  that may or may not be aligned with  $S_c$ . The two directories vary in the following manner. When a new user commit is proposed, we use `rsync` tool [23] to efficiently synchronize  $S_c$  and  $S_d$ , thereby ensuring that it is clean; we then copy all new `Mizar` source files to the dirty sandbox (that was just cleaned). Note that using `rsync` for this task provides a reasonable trade-off for solving several different concerns:

---

<sup>13</sup> We are making use of the fact that in each verifiable `Mizar` article, one must declare what kinds of notations, definitions, proofs, etc., one depends on. If `Mizar` did not have this feature, then calculating dependencies would, presumably, be more difficult. Also note that these are just dependencies between articles, while it is certainly interesting future work to also calculate the precise dependencies between smaller-scale article *items* and use this information for smarter and leaner re-verification. The MPTP system [24] can be used for extracting information of this kind.

- to check the newly arrived changes without possibly destroying the previous correct version, we need to have a fresh directory with the most recent correct version;
- the directory should contain as much pre-computed information as possible, because re-verifying and HTML-izing the whole library (more than 1000 articles) from scratch is an expensive operation (taking hours, even with high-end equipment), while we want to be as real-time as possible and re-use all available precomputed information;
- while the directory containing just the *Mizar* articles is reasonably small (81MB at the moment), the directory with the complete pre-computed information is prohibitively big: the internal library created from the *Mizar* articles, the environment for each article, and the HTML files are together more than 10GB.<sup>14</sup> Simply copying all this pre-compiled information would rule out a real-time experience, especially in cases when the user wants to change an article on which not many other articles depend.

Using `rsync` into a fresh directory addresses the first two issues (keeping the clean install intact and not re-verifying all from scratch). Using `rsync`, on average, reasonably ensures (by relying on smart `rsync` mechanisms) that of the clean sandbox over the dirty one does not take too long.

Since the clean sandbox  $S_c$  contained *everything*—*Mizar* source files and all generated auxiliary files—the dirty sandbox  $S_d$  now contains (after adding the newly modified *Mizar* files) nearly everything, too. All that is missing are up-to-date auxiliary files<sup>15</sup> to be generated from the modified *Mizar* source files. For that we invoke the master makefile  $M$  to generate possibly new dependency makefiles  $M_a$ , and then we invoke the master makefile to request a re-build/re-verification of the entire MML. Since we are working in a sandbox  $S_d$  that contains the all results of previous builds of the entire MML, “most” of the *Mizar* source files will not need to be looked at. (We put “most” in quotes because if one proposes to commit an update to a sufficiently basic *Mizar* article, then a good deal of computation is required to verify and propagate the change over the whole library, even though it could actually be a rather small edit. But such “foundational” edits are, apparently, uncommon.)

By using makefiles this way we are also able to exploit their ability to run multiple jobs in parallel. The dependency graph of the full MML is wide and rich enough that, when making “normal” edits, we can indeed benefit from this feature of make when running on multi-core machines or when employing grid/cloud-computing. Indeed, this is crucial for us to provide a sufficiently quick response to users who submit “normal” changes to MML.

---

<sup>14</sup> This blow-up is caused by creating a local (XML) environment files for every article, and by having detailed XML and HTML representations of the article. A lot of information is kept in such files in a verbose form.

<sup>15</sup> That is, files witnessing the correctness of the verification, updated library files, re-generated HTML files, etc.

Further opportunities for parallelization, based on a finer-grained analysis of Mizar articles than the one discussed here (where the dependency relation is between an entire Mizar article on other articles), are being investigated.

## 7 Future Work and Summary

A number of features can be added to our prototype. As already discussed, the world of wikis is dynamic and it is likely that all kinds of web frameworks and technologies will be used in the future for presenting the Mizar library, and collaboratively growing it. This is also why we stress a good basic model for dealing with the main data (articles) and processes (collaboration), i.e., using Git as a good mainstream DVCS, with a rapidly growing number of tools, frontends, and public collaborative platforms based on it.

The features that are being implemented at the time of writing this paper include: basic web-based editing; finer itemization, dependencies, parallelization and thus verification and HTML-ization speed for both simple (one-article) commits and multi-article commits; plugging in all kinds of powerful proof advice tools (automated theorem provers, library search engines, AI systems, etc.). Obviously these require nontrivial effort on all these tools and interfaces to them. Nontrivial and continuing work is actually already providing good HTML presentation of the formal articles, with all kinds of additional useful information and functions (following the symbols to their definitions being the most obvious one).

We have already mentioned the variety of tools available for Git and how public collaborative sites like GitHub rapidly develop all kinds of collaborative functions on top of Git. A similar remark is also true about, e.g., ikiwiki, and possibly about other wikis based on DVCSs. Thus, it seems to us that the mission of formal mathematical wiki builders should be to watch these rapidly developing collaborative technologies, customizing them (by providing suitable commit hooks, HTML-izers, dependency utilities, change propagation models, etc.) and complementing them (by providing all kinds of support tools developed for formal mathematics) rather than competing with them by starting from scratch.

While this paper is mainly about the Mizar proof assistant and its library, it should be clear that the functionalities we discussed (tools for good dependency extraction and focused re-verification and HTML-ization, possibly itemization and parallelization, all kinds of useful proof advice tools, etc.), the ideas and work presented here could be instantiated also to other proof assistants (Coq, Isabelle, HOL, etc.). When this is done, formal wikis could become the place where various formalisms and alternative formalizations meet, allowing collaboration on large formal projects, possibly having mechanisms also for gradual formalization of informal mathematical texts, and also allowing formal texts that are not completely correct. Our hope is that this infrastructure will attract more “normal/traditional” mathematicians to the advantages of formal mathematics, gradually making it more digestible, and possibly thus allowing further important steps in the inevitable computerization of human mathematics.

## References

1. The Archive of Formal Proofs, <http://afp.sourceforge.net/>
2. Wikiwikiweb, <http://c2.com/cgi/wiki?WikiWikiWeb>
3. Corbineau, P., Kaliszyk, C.: Cooperative repositories for formal proofs. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) *Calculemus/MKM. Lecture Notes in Computer Science*, vol. 4573, pp. 221–234. Springer (2007)
4. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN, the Constructive coq Repository at Nijmegen. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) *Mathematical Knowledge Management, Third International Conference, MKM 2004, Białowieża, Poland, September 19–21, 2004, Proceedings*. vol. 3119, pp. 88–103. Springer (2004)
5. D’Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 27(7), 1165–1178 (2008)
6. Git - fast version control system, <http://git-scm.com/>
7. Gonthier, G.: Formal proof—the four-color theorem. *Notices of the American Mathematical Society* 55(11), 1382–1393 (2008)
8. Hales, T.: A proof of the Kepler conjecture. *Annals of Mathematics* 162(3), 1065–1185 (2005)
9. Hales, T.: Formal proof. *Notices of the American Mathematical Society* 55(11), 1370–1381 (2008)
10. Harrison, J.: Formal Proof – Theory and Practice. *Notices of the American Mathematical Society* 55, 1395–1406 (2008)
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: *seL4: Formal verification of an OS kernel*. In: Anderson, T. (ed.) *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. pp. 207–220. ACM Press (2009)
12. Mathoverflow, <http://mathoverflow.net/>
13. Wolfram MathWorld: The Web’s Most Extensive Mathematics Resource, <http://mathworld.wolfram.com/>
14. Matuszewski, R., Rudnicki, P.: MIZAR: the first 30 years. *Mechanized Mathematics and its Applications* 4(1), 3–24 (2005)
15. The Mizar Mathematical Library, <http://mizar.org/library/>
16. Naumowicz, A., Kornilowicz, A.: A Brief Overview of Mizar. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs. Lecture Notes in Computer Science*, vol. 5674, pp. 67–72. Springer (2009)
17. Planetmath, <http://planetmath.org/>
18. The polymath blog, <http://polymathprojects.org/>
19. Proofwiki, [http://www.proofwiki.org/wiki/Main\\_Page](http://www.proofwiki.org/wiki/Main_Page)
20. The QED Manifesto. In: *Automated Deduction - CADE 12. Lecture Notes in Artificial Intelligence*, vol. 814, pp. 238–251. Springer-Verlag (1994)
21. The QPQ Deductive Software Repository, <http://www.qpq.org>
22. Rudnicki, P., Trybulec, A.: On the integrity of a repository of formalized mathematics. In: *MKM. Lecture Notes in Computer Science*, vol. 2594, pp. 162–174. Springer (2003)
23. Trigdell, A.: *Efficient Algorithms for Sorting and Synchronization*. Ph.D. thesis, Australian National University (1999)
24. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning* 37(1-2), 21–43 (2006)
25. vdash: What is vdash?, <http://www.vdash.org/intro/>